Principles of NoSQL Databases

BDNR · Non-Relational Databases
M.EIC · Master in Informatics Engineering and Computation

Sérgio Nunes
Dept. Informatics Engineering
FEUP · U.Porto

Based on Chapter 1 from Next Generation Databases, Harrison (2016)
Based on Chapters 2, 3, 4, 5, and 7 from NoSQL Distilled, Sadalage and Fowler (2012)

Agenda

- Evolution of Database Management Systems
  - Early Systems
  - Relational Database Systems
  - Motivations for NoSQL
- Principles of NoSQL Databases
  - Aggregate Data Models
  - Data Distribution Models
  - Data Consistency
- Distributed Data Processing
  - Map-Reduce

Evolution of Database Management Systems

Based on "Chapter 1: Three Database Revolutions" from Next Generation Databases, Harrison (2016)

Three Database Revolutions

- The first database revolution was driven by the electronic computer.
  - Pillars: database management systems; record at a time processing, flat files.
- The second revolution is associated with the relational database paradigm.
  - Pillars: relational model; ACID transactions; and the SQL language.
- The third (current) revolution is associated with NoSQL solutions.
  - Pillars: non-relational; large scale; distributed and global scope.

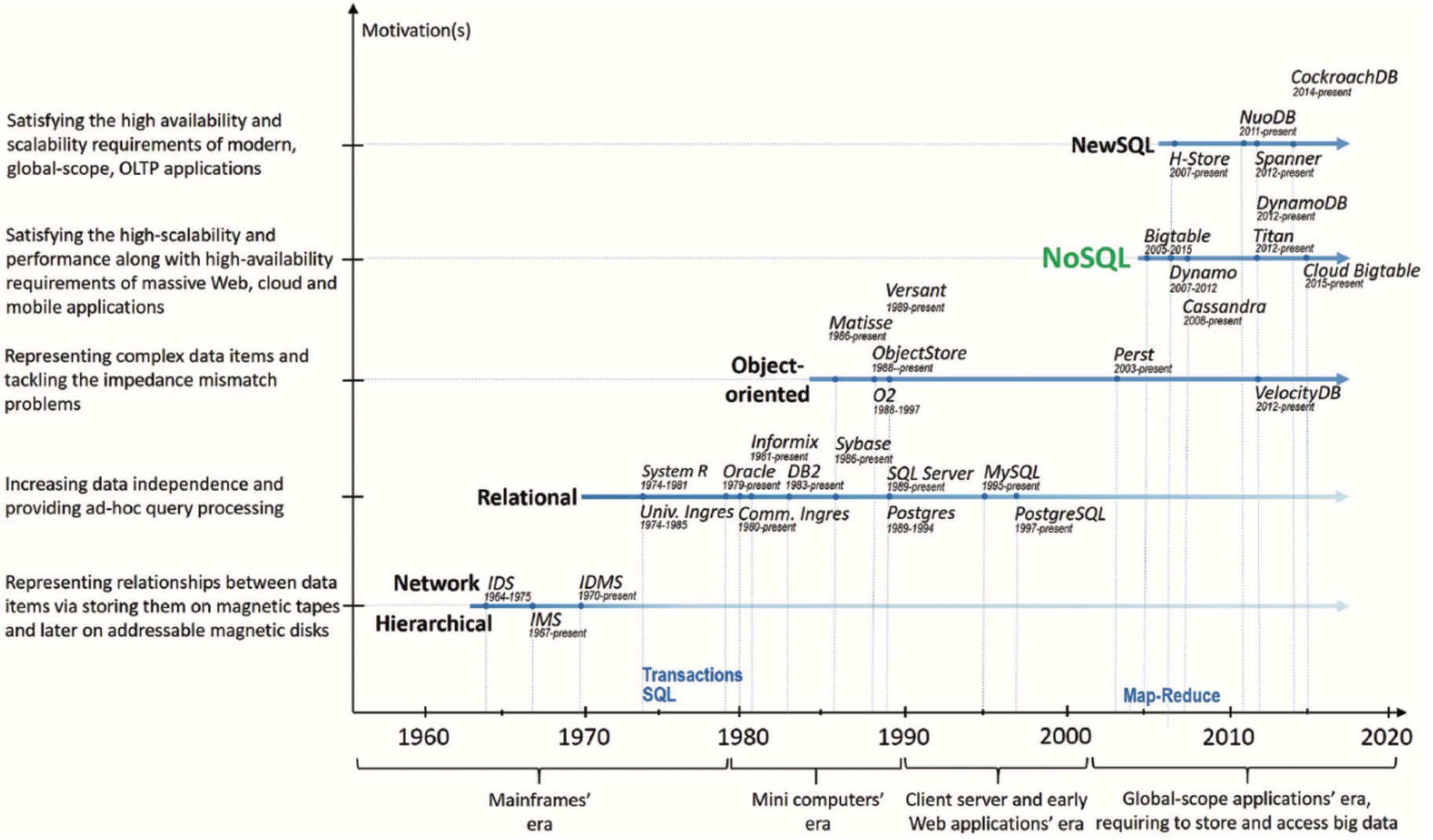

Fig. 1. The continuous development of major database technologies and some corresponding database systems.

Image from A Survey on NoSQL Stores, Davodian et al. (2018)

First Database Revolution

First Database Revolution

- Data access and manipulation was incorporated in the application code, which lead to multiple problems: reinventing the wheel, duplicated code, risk of problems in application code, and required expertise for data management development.
- Database management systems became separate from the application.
- The first generation of DBMS were dominated by two models:
  - Hierarchical model, where all data is represented as a tree of records nested within records (similar to what is found in JSON today).
  - Network model, a generalization of the hierarchical model with the extension that a node could have multiple parents.
- These systems were described as navigational, since it was required to navigate from one object to others using pointers or links.

Hierarchical Model and Network Model

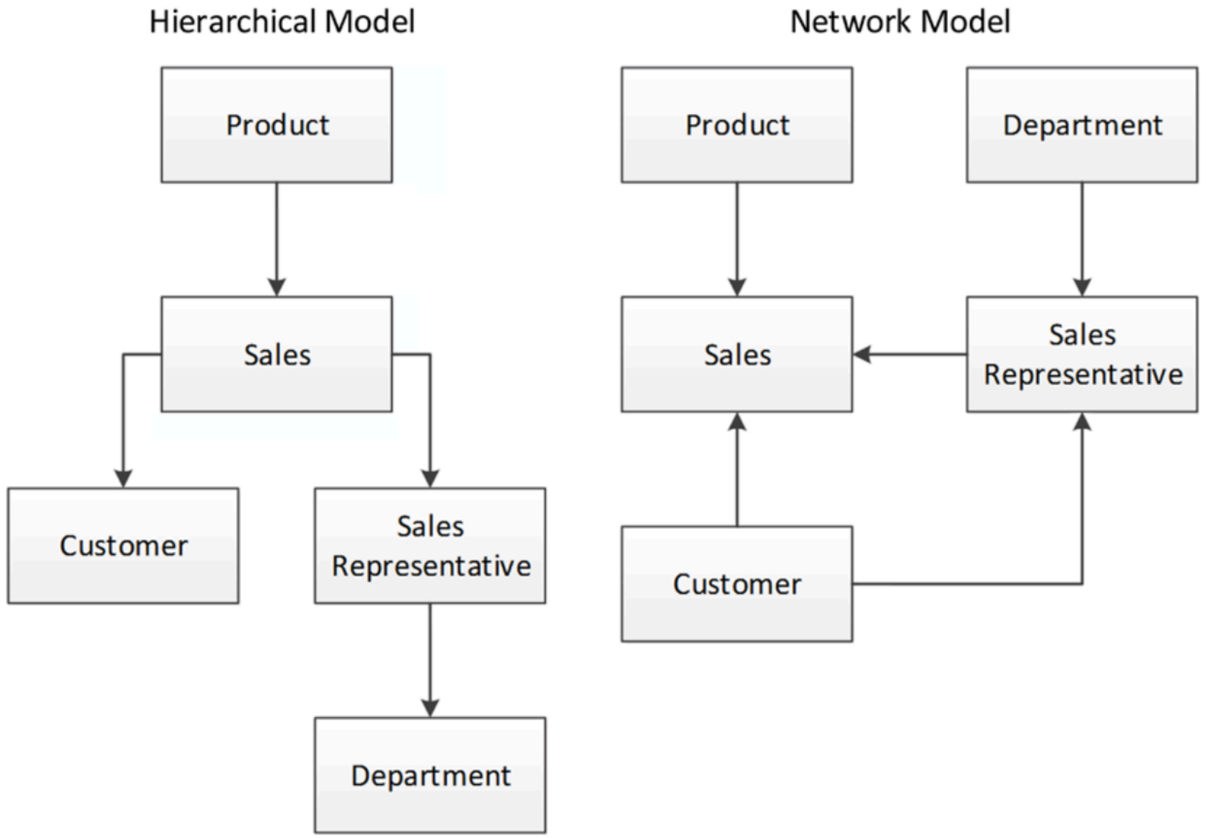

Figure 1-3. *Hierarchical and network database models*

Image from Next Generation Databases, Harrison (2016)

Second Database Revolution

Second Database Revolution

- In 1960s, Edgar Codd, identified three major limitation in previous databases:
  - Too hard to use, i.e. only accessible to people with specialized programming skills;
  - Lacked a theoretical foundation, i.e. arbitrary representations that did not ensure logical consistency or provide the ability to deal with missing information;
  - Mixed logical and physical implementations, i.e. the representation of data matched the format of the physical storage, rather than the logical representation of the data.
- He proposed (the now standard) relational database model, i.e. putting all data in relations, which are collections of tuples.
- Establishing the foundations to Relational Database Management Systems (RDBMS).

Relational Model

- Recall the main concepts of the relational model:
  - Tuples, are an unordered set of attribute values.
  - Relations, are a collection of distinct tuples.
  - Constraints, enforce consistency of the database.
    - Key constraints are used to identify tuples and relationships between tuples.
  - Operations on relations such as selections and projections, return relations.
- Levels of conformance to the relational are described in the various "normal forms", designed to avoid anomalies resulting from data manipulation (inserts, updates, deletes).

Data Normalization

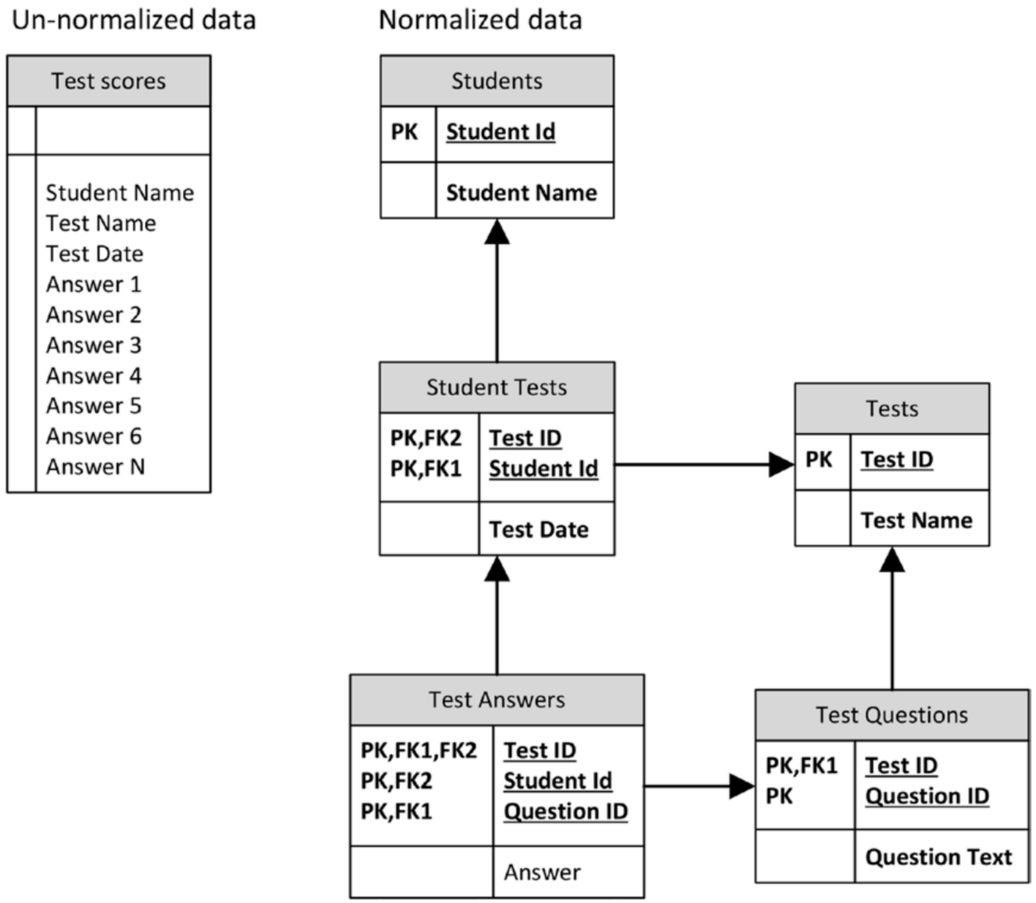

Figure 1-4. *Normalized and un-normalized data*

Image from Next Generation Databases, Harrison (2016)

Transaction Models

- The relational model does not define how concurrent changes are handled.
- Transactions encapsulate these changes. Jim Gray' definition:
  - A transaction is a transformation of state which has the properties of atomicity (all or nothing), durability (effects service failures), and consistency (a correct transformation).
- Popularized as ACID transactions, which became the standard in implementations.
  - Atomic, the transaction is indivisible (all or none statements are applied);
  - Consistent, the database remains in a consistent state after the transaction;
  - Isolated, one transaction should not affect other in-progress transactions;
  - Durable, once a transaction is saved its changes are expected to persist even if there is a failure to the operating system or hardware.

Object-Oriented Database Management Systems

- The object-oriented programming paradigm challenged traditional procedural programming languages. Representation and behavior was merged into a single object.
- The impedance mismatch, i.e. the effort required to map between complex data structures and their relational representation, led to efforts in developing object-oriented databases (mid 1990s).
- An Object Oriented Database Management System (OODBMS) would store directly objects without normalization and would allow to load and store objects easily.
- However, these solutions failed to gain market adoption. One of the reasons was the lack of a non-programmer interface, that enabled business users to query databases (such as SQL).

Mismatch Between Representations

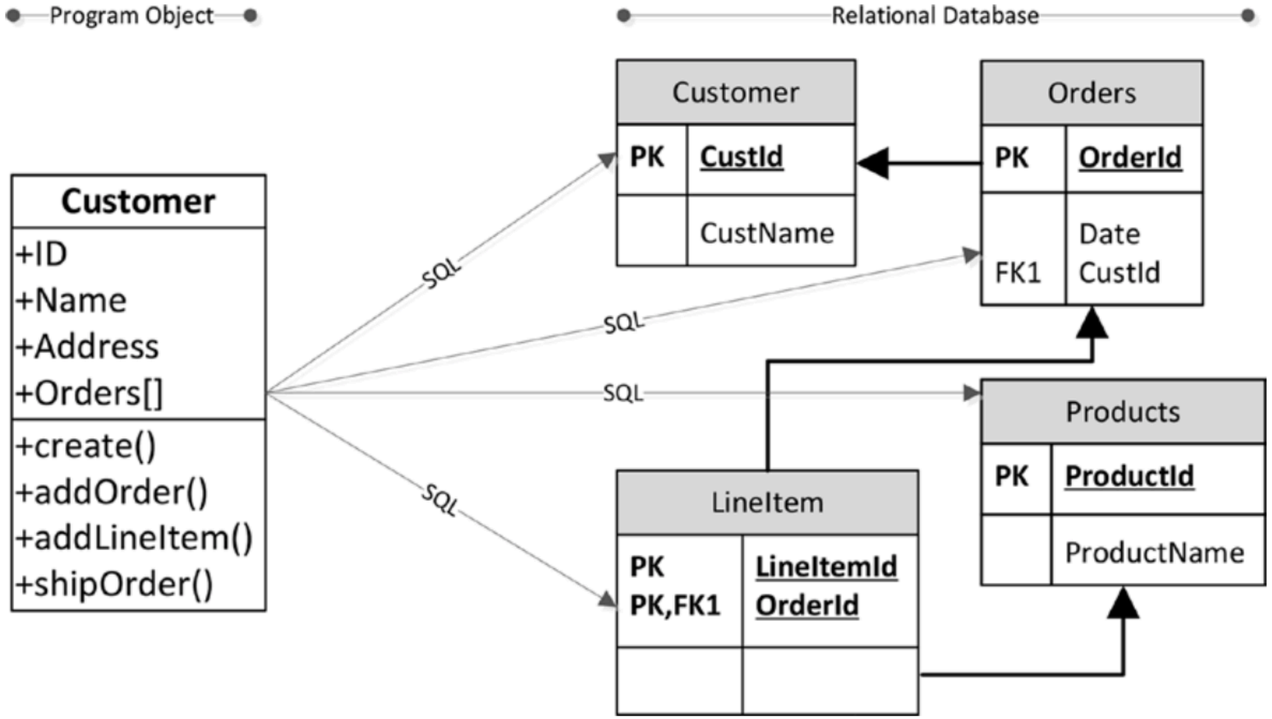

Figure 1-5. *Storing an object in an RDBMS requires multiple SQL operations*

Image from Next Generation Databases, Harrison (2016)

Third Database Revolution

Web Scale and Cloud Computing

- Google was pioneer in the development of new solutions to handle growing data volumes, both in storage and processing. Innovations from the early 2000s include, the Google File System, Map-Reduce framework, and BigTable.

- Centralized solutions based on RDBMS were expensive to license and maintain. This lead to the adoption of techniques such as sharding and in-memory distributed data storage solutions.

- In the late 2000s, infrastructure as a service emerged and accelerated the development of new distributed solutions for storage and computation, with Amazon leading this evolution.

Document Databases

- Serialized representation of objects, in XML or JSON, in the context of message-based interactions (e.g. AJAX), were increasingly adopted by programmers.
- Document-based representations reduce the burden of translating objects from and to the relational model (see impedance mismatch).
- JSON objects can be stored in columns within relational databases, but native document-based databases appeared, such as CouchBase and MongoDB, where documents are the central concepts.

Motivations for NoSQL

- Scalability
  - In centralized RDBMS is difficult to scale out (more machines) and scaling up (better machine) is expensive and has limits.
  - NoSQL solutions are designed to be implemented in clusters, thus scaling out is supported by design.
- Cost
  - Multiple licensing formulas exist (by users, cores, etc), none of which adequate for varying loads (e.g. web).
  - The majority of NoSQL solutions are open-source, avoiding these issues.
- Flexibility
  - Relational databases require strong investments in the initial design and changes to the schema has costs.
  - Some NoSQL databases are schemaless in the sense that they do not require fixed structures.
- Availability
  - Centralized RDBMS typically have single points of failure, thus demand higher infrastructure investments.
  - Most NoSQL database are designed to take advantage of multiple low-cost servers, and adapt by design to failures.

Summary of the Evolution of Database Management Systems

- The first database revolution was a result of electronic digital computers.
  - The mainframe was the platform.
- The second database revolution resulted from development of the relational model. The relational database evolved and consolidated itself across more than 30 years of commercial dominance.
  - The platform was centered on the client-server model, with centralized RDBMS.
- The third database revolution is centered on scaling from single node architectures to deal with the size in data volume and number of users.
  - The platform is distributed and multiple solutions for data storage exist, that include but are not limited to RDBMS — polyglot persistence.

Principles of NoSQL Databases

Aggregate Data Models

Based on "Chapter 2: Aggregate Data Models" from NoSQL Distilled, Sadalage and Fowler (2012)

Data Models

- A data model is the model through which we perceive and manipulate our data.
- The data model describes how a database user interacts with the data.
- This is distinct from a storage model, which describes how data is stored internally.
- Also distinct from the conceptual data model, which describes the high-level concepts and relationships used to describe a domain.
- The relational data model is the dominant data model, which includes the concepts of tables (relations), columns (attributed) and rows (tuples).
- The NoSQL ecosystem is dominated by four main categories of data models: key-value, document, column / tabular, and graph.

Aggregates

- The relational model divides the information into tuples (rows).
- A tuple is a limited data structure:
  - It captures a set of literal values;
  - It does not support nested tuples (records);
  - It is not possible to use lists as tuple values.
- Aggregate orientation takes a different approach, recognizing that is often necessary to operate on data in units that have a more complex structure than a set of tuples.
  - Example: a complex data record that allow list and other data record structures to be nested inside it.
- The term "aggregates" refers to these complex records, i.e. a collection of related objects that are treated as a unit. It corresponds to a unit for data manipulation and management of consistency.
- Typically, "aggregate structures" are easier for programmers to work with (e.g. retrieve, update).

Example of Relations and Aggregates

- Consider an e-commerce website, where it is necessary to store information about the product catalog, orders, shipping and billing addresses, and payment data.

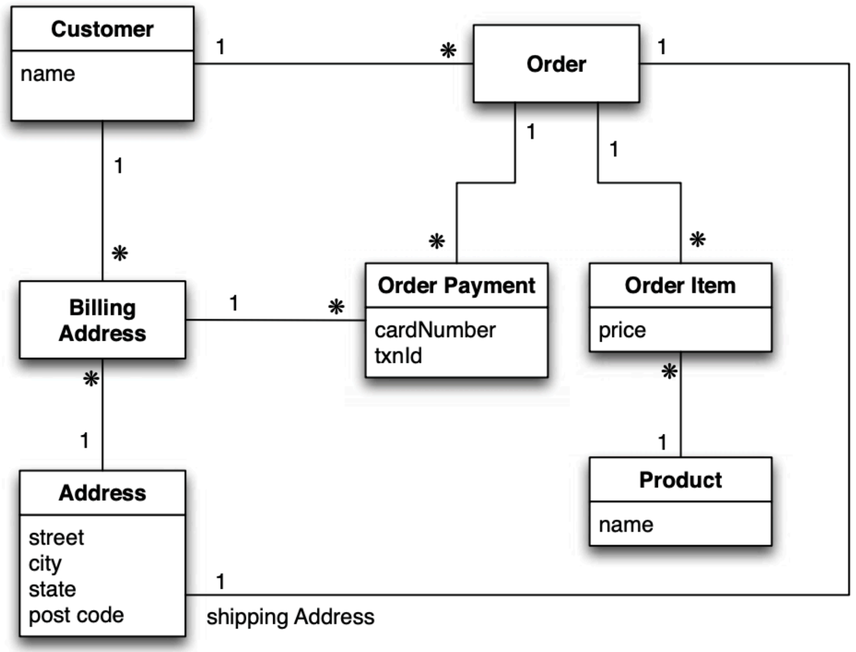

Figure 2.1 *Data model oriented around a relational database (using UML notation [Fowler UML])*

Customer

Id	Name
1	Martin

Order

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Figure 2.2 *Typical data using RDBMS data model*

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Example Aggregate Data Model

- The same model in more aggregate-oriented terms. UML compositions (black-diamond marker) are used to show that data fits into the aggregation structure.

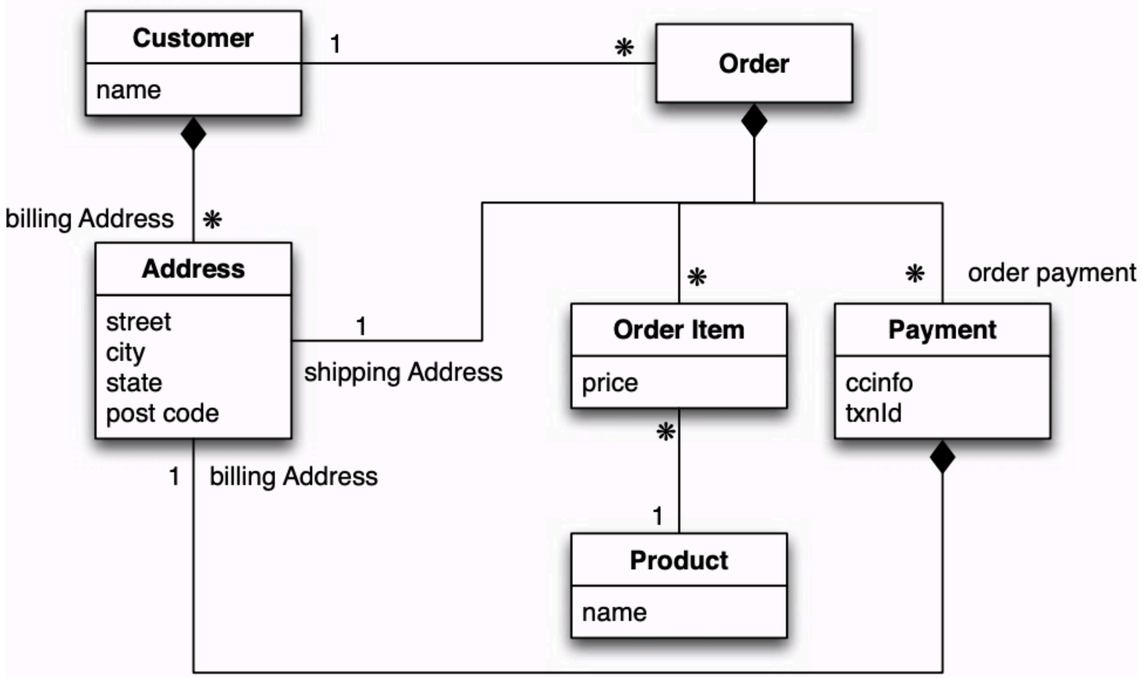

Figure 2.3 *An aggregate data model*

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Example Aggregate Data Model (2)

- A single logical address record appears three times in the example data, but instead of using IDs it is treated as a value and copied each time.
- With aggregates, we can copy the whole address structure into the aggregate as we need to.
- The product name is included as part of the order item here — this kind of denormalization is similar to the tradeoffs with relational databases, but is more common with aggregates because the goal is to minimize the number of aggregates accessed during a data interaction.
- The important thing to notice isn't the particular way the aggregate boundary are drawn, so much as the fact of how data is accessed — and make that part of the processes when developing the application data model.

Example Alternative Aggregate Data Model

- Aggregate boundaries could be drawn differently, putting all the orders for a customer into the customer aggregate.

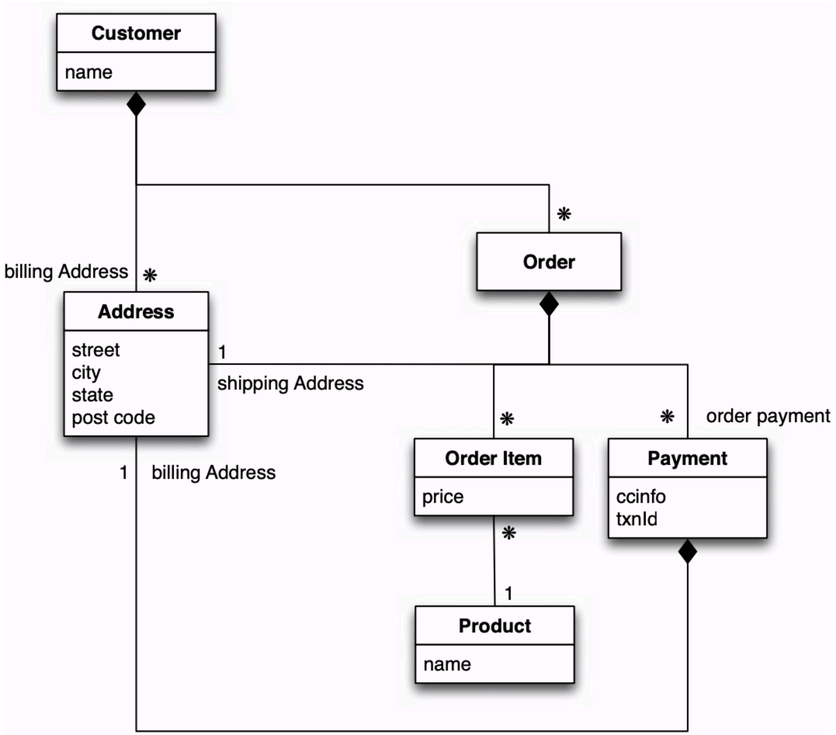

Figure 2.4 *Embed all the objects for customer and the customer's orders*

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]

  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
  }]
}
}
```

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Impact of Aggregate-Oriented Models

- The relational mapping captures the various data elements and their relationships reasonably well, but it does so without any notion of an aggregate entity.
- In the relational model, relationships can be expressed in terms of foreign key relationships, but there is nothing to distinguish relationships that represent aggregations from those that don’t.
- Thus, the database can’t use knowledge of aggregate structure to help it store and distribute the data.
- When working with aggregate-oriented databases, there is a clearer semantics to consider by focusing on the unit of interaction with the data storage. It is, however, not a logical data property: it’s all about how the data is being used by applications—a concern that is often outside the bounds of data modeling.
- Relational databases have no concept of aggregate within their data model, they're labeled as aggregate-ignorant. In the NoSQL world, graph databases are also aggregate-ignorant.

Aggregate-Ignorant Models

- Being aggregate-ignorant is a characteristic, not a problem.
- It's often difficult to draw aggregate boundaries well, particularly if the same data is used in many different contexts.
  - An order makes a good aggregate when a customer is making and reviewing orders, and when the retailer is processing orders.
  - However, if a retailer wants to analyze its product sales over the last few months, then an order aggregate becomes a trouble. To get to product sales history, you'll have to dig into every aggregate in the database.
- An aggregate structure may help with some data interactions but be an obstacle for others.
- An aggregate-ignorant model allows you to easily look at the data in different ways, so it is a better choice when you don't have a primary structure for manipulating your data.

Transactions in Aggregate-Oriented Models

- A strong reason for aggregate orientation is that it helps greatly with running on a cluster (one of the main arguments for the rise of NoSQL).
- When running on a cluster, there is a need to minimize the number of nodes queried to gather data. By explicitly including aggregates, the database is given important information about which bits of data will be manipulated together, and thus should live on the same node.
- Aggregates have an important impact for transactions. With atomicity, rows spawning many tables are updated as a single operation that either succeed or fail it its entirety.
- In general, aggregate-oriented databases don't have ACID transactions that span multiple aggregates. Instead, they support atomic manipulation of a single aggregate at a time.
- Handling multiple aggregates in an atomic way needs to be managed in the application code.

Key-Value and Document Data Models

- Key-value and document database are strongly aggregate-oriented, i.e. they are primarily constructed through aggregates. Both database types consists of many aggregates with each aggregate having an access key or ID.
- The difference between the two models is that
  - in key-value model, the aggregate is opaque to the database;
  - while in the document model, the database sees the structure in the aggregate.
- With a key-value store, an aggregate can only be accessed by lookup based on its key.
- With a document database, queries can use the fields in the aggregate for lookup; parts of the aggregate can be retrieved rather than the whole thing; and indexes can be created based on the contents of the aggregate.
- In practice, database systems combine features from both models.

Column-Oriented Databases

- In column-oriented databases, the unit of storage are the columns, in contrast with the relational approach based on grouping by rows.

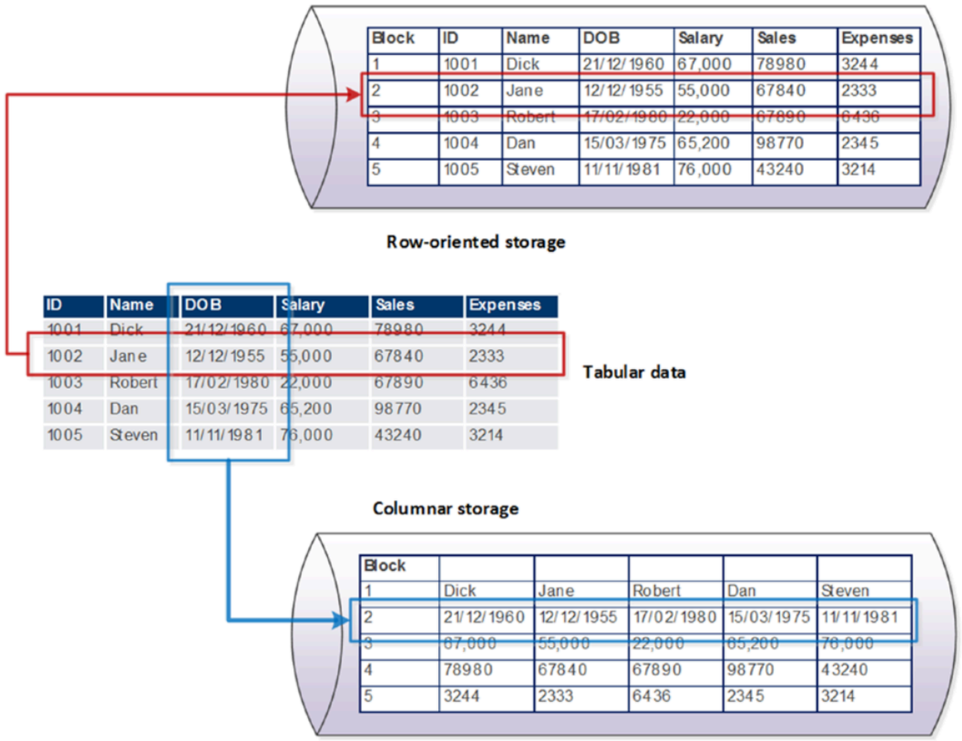

Figure 6-2. *Comparison of columnar and row-oriented storage*

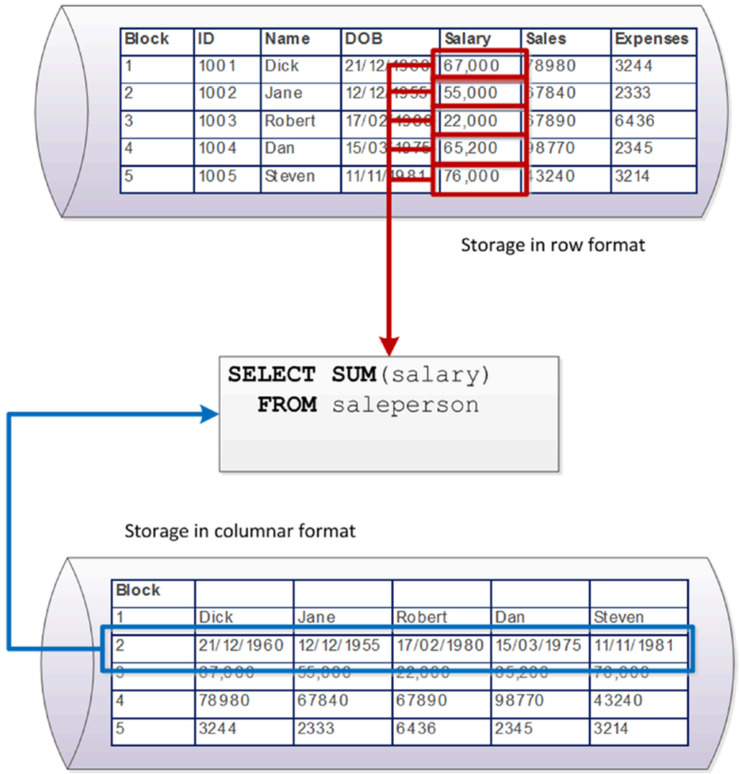

Figure 6-3. *Aggregate operations in columnar stores require fewer IOs*

Images from Next Generation Databases, Harrison (2016)

Column-Oriented Databases

- Having rows as a unit of storage helps write performance. In scenarios where writes are rare, but read a few columns of many rows at once, it's better to store groups of columns for all rows as the basic storage unit.
- The column acts as the unit for access, with the assumption that data for a particular column family will be usually accessed together.
- Row-oriented: each row is an aggregate (e.g. customer with ID 123) with column families representing data (e.g. profile, order history) within that aggregate.
- Column-oriented: each column family defines a record type (e.g. customer profiles) with rows for each of the individual records. Column families can contain a single column (skinny) or many columns (wide).
- Since the database knows about this grouping of data, it can use this information to define storage and access behavior.

Summary of Aggregate-Oriented Models

- Aggregate-oriented models share the notion of an aggregate (complex unit of data) indexed by a key that can be used for lookup.
- This aggregate is central to running a distributed system, as the database will ensure that all the data for an aggregate is stored together on one node.
- The key-value model treats the aggregate as an opaque whole, only key-based lookups are possible.
- The document model makes the aggregate transparent to the database, allowing partial retrievals and lookups based on aggregate values.
- The column-oriented model divides the aggregate into column families, allowing the database to treat them as units of data within the row aggregate.

Summary of Aggregate Data Models

- An aggregate is a collection of data that is manipulated as a unit.
- Aggregates form the boundaries for ACID operations with the database.
- Key-value, document, and column-oriented database are forms of aggregate-oriented databases.
- Aggregates make it easier for the database to manage data storage over clusters.
- Aggregate-oriented databases work best when most data interaction is done with the same aggregate.
- Aggregate-ignorant databases are better when interactions use data organized in many different formations.

Distribution Models

Based on "Chapter 4: Distributed Models" from NoSQL Distilled, Sadalage and Fowler (2012)

Distribution Models

- A central driver for NoSQL technologies is its ability to run on a distributed fashion.
- As data volumes increase, it becomes more difficult and expensive to scale up, i.e. invest in a bigger server to run the database on.
- A mode appealing option is to scale out, i.e. run the database on a cluster of servers.
- Aggregate-oriented models fit well with scaling out because the aggregate is a natural unit for distribution.
- Depending on the distribution model, a data store can handle larger quantities of data, processes greater read or write traffic, or improve availability in the face of failures.
- There are two paths to data distribution: replication and sharding.

Single Server

- The simplest, and most common option, is to have no distribution.
- The database is run on a single machine that handles all the reads and writes.
- This approach eliminates all the complexities that arise with the distributed options, it is easier to manage and to use for application developers.
- Using NoSQL in a single-server may make sense if the data model of the NoSQL database is more suited to the application.
  - Graph databases in particular work best in a single-server configuration.
- If distribution is not necessary, a single-server approach is generally the best option.

Sharding

- Sharding is a data distribution technique that supports horizontal scalability, where different parts of the data are put into different servers.
- This is suitable for "disjoint workloads", i.e. different people typically access different parts of the dataset.

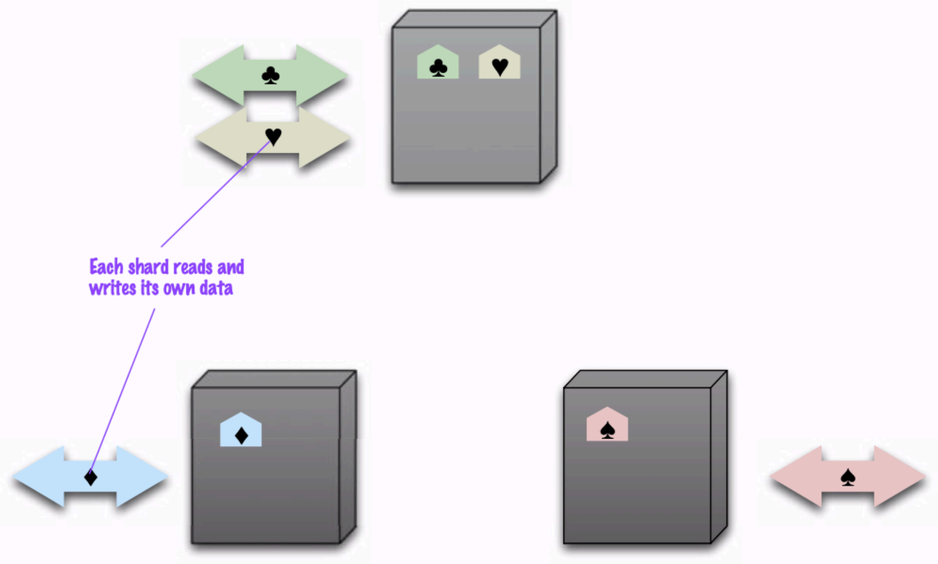

Figure 4.1 *Sharding puts different data on separate nodes, each of which does its own reads and writes.*

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Sharding Partitioning

- When using sharding, the first question is how to cluster data between nodes so that cross-node accesses are minimized.
- Several factor can impact this decision: physical location (e.g. organize data by geographical location of users); temporal access patterns (e.g. historical versus current data); logical access patterns (e.g. alphabetic customer name order).
- Sharding improves both read and write performance.
- Sharding alone has little impact on improving resilience. Although the data is on different nodes, there is a single copy of each shard. Failures have a localized impact but the impact is definitive on the affected data, and more "single-points of failure" exist.
- Sharding is made easier with aggregates.
- Moving from single node to sharding requires planning and can be tricky.

Sharding Example

- Sharding example with two distinct strategies:
  - Range-based, each shard is allocated a specific range of shard key values.
  - Hash-based, the keys are distributed based on a hash function applied to the shard key.
- Range-based partitioning allows for more efficient execution of queries that process ranges of values, since these queries can often be resolved by accessing a single shard.
- Hash-based sharding requires that range queries be resolved by accessing all shards. On the other hand, hash-based sharding is more likely to distribute “hot” documents (e.g. recent posts) evenly across the cluster, thus balancing the load more effectively.

Example from Next Generation Databases, Harrison (2016)

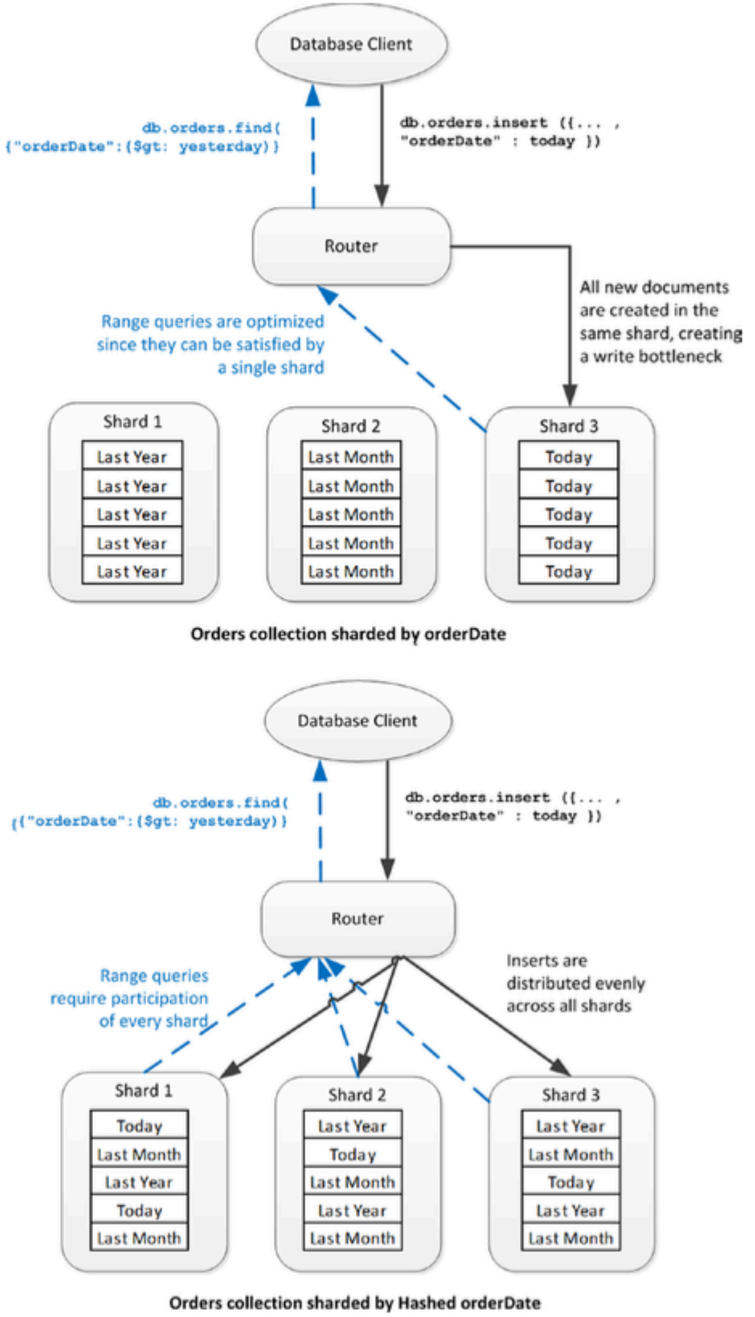

Figure 8-6. *Comparison of range and hash sharding in MongoDB*

Primary-Replica* Distribution

- With primary-replica distribution, data is replicated across multiple nodes.
- One node is designated the primary, and is the authoritative source for the data and usually responsible for processing updates to that data.
- The other nodes are the replicas, or secondary nodes.
- Most adequate for "read-intensive" scenarios.
- Horizontal scalability is handled by adding more replica nodes and ensuring that all read requests are routed to the replicas.
- For heavy write traffic, it is not a good scheme.

* Sometimes referred to as "Master-Slave" but "Words Matter" [https://www.acm.org/diversity-inclusion/words-matter].

Primary-Replica Distribution

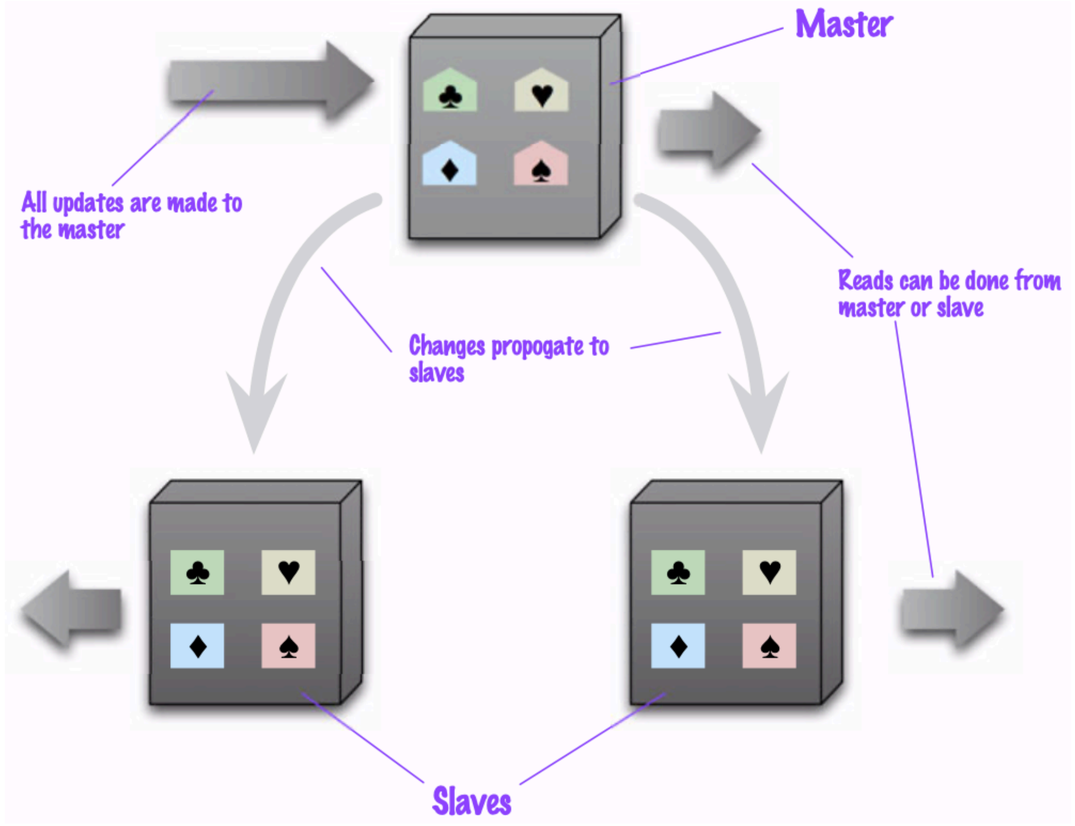

Figure 4.2 *Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.*

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Primary-Replica Resilience

- This scheme improves read resilience:
  - If the primary fails, read operations can still be handled by the replicas.
  - Although write requests cannot be answered.
- Primary recovery is made easier since a replica can be appointed as primary.

- Important risk of this pattern is inconsistency:
  - Different clients reading from different replicas and seeing different values because changes haven't propagated to all replicas.

Peer-to-Peer Replication

- In primary-replica distribution, the primary node is a bottleneck and a single point of failure.
- With peer-to-peer replication these problems are addressed by not having a primary node.
- All replicas have equal responsibilities, they can accept both reads and writes.
- The most important drawback is consistency.
  - When is possible to write to two different places there is a risk of having update attempts to the same record at the same time, resulting in a write-write conflict.
  - Avoiding these conflicts requires more coordination.

Peer-to-Peer Replication

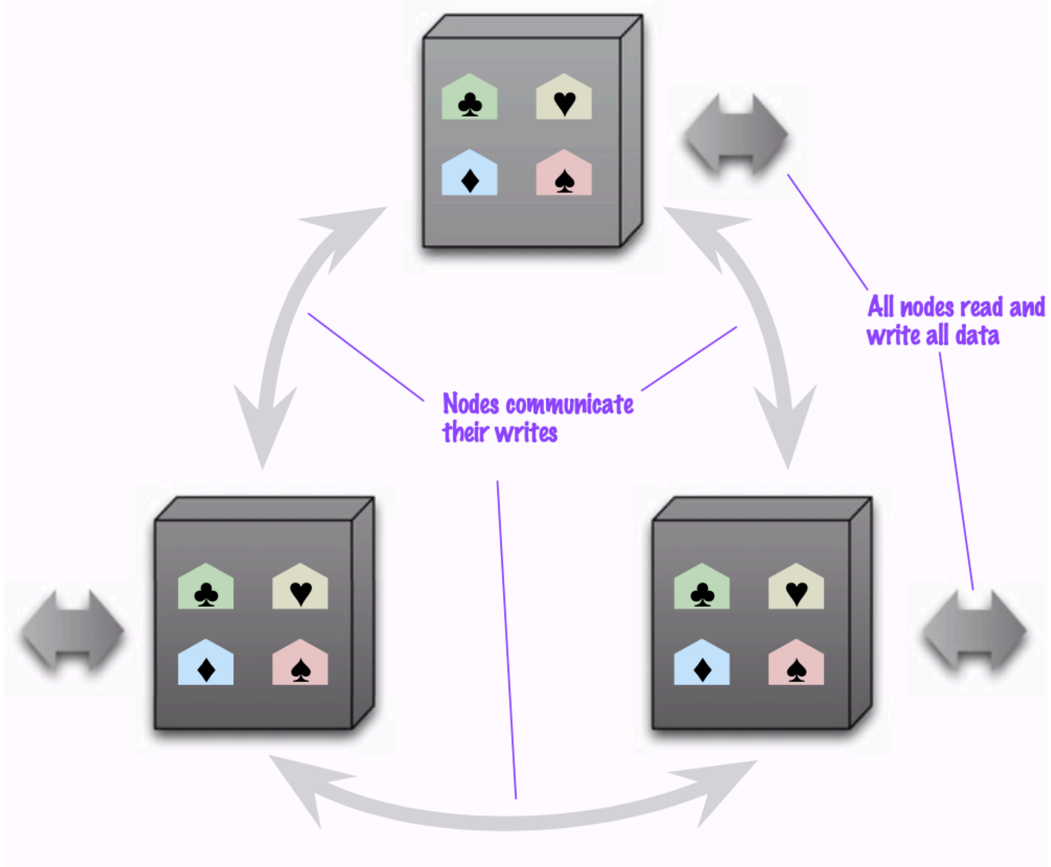

Figure 4.3 *Peer-to-peer replication has all nodes applying reads and writes to all the data.*

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Combining Sharding and Replication

- Replication and sharding strategies can be combined.
  - With sharding and primary-replica distribution, multiple primaries exist, but each data item only exists on a single primary.
  - With sharding and peer-to-peer, each shard is replicated in multiple nodes (e.g. 3).

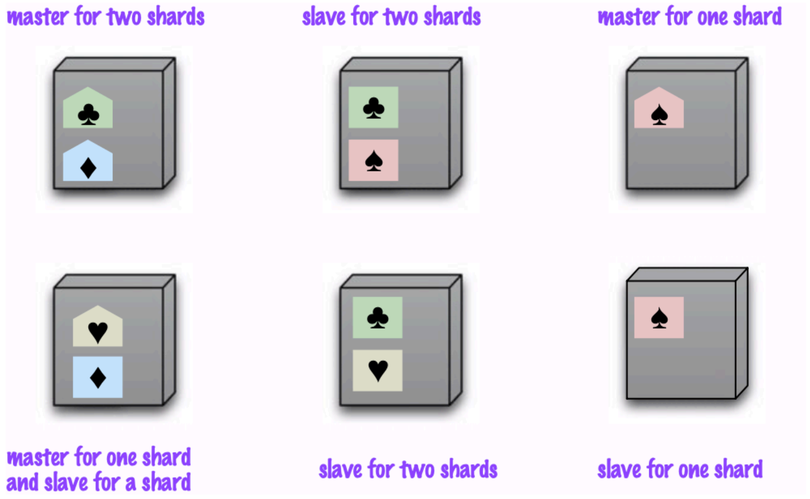

Figure 4.4 *Using master-slave replication together with sharding*

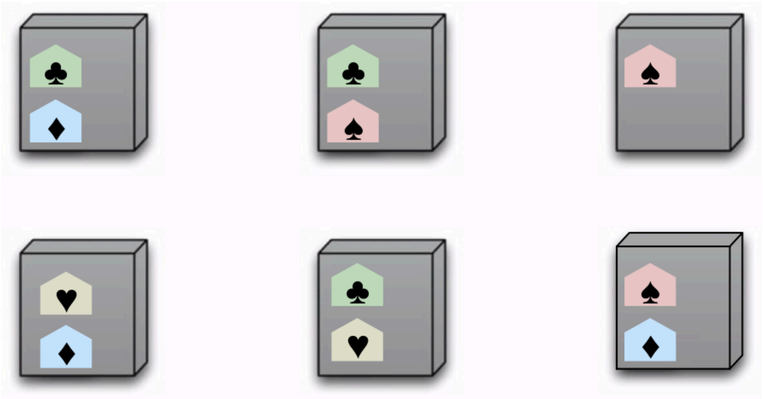

Figure 4.5 *Using peer-to-peer replication together with sharding*

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Summary of Distribution Models

- There are two styles of distributing data (a system may use either or both):
  - Sharding distributes different data across multiple servers, so each server acts as the single source for a subset of data.
  - Replication copies data across multiple servers, so each bit of data can be found in multiple places.
- Replication comes in two forms:
  - Primary-replica distribution makes one node the authoritative copy that handles writes, while replicas synchronize with the primary and may handle reads.
  - Peer-to-peer replication allows writes to any node; the nodes coordinate to synchronize their copies of the data.
- Primary-replica approach reduces the changes of update conflicts, but peer-to-peer replication avoids loading all writes onto a single point of failure.

Data Consistency

Based on "Chapter 5: Consistency" from NoSQL Distilled, Sadalage and Fowler (2012)

Data Consistency

- Centralized relational databases try to achieve strong consistency.
- Distributed NoSQL databases changes the way consistency is viewed.
- Next topics:
  - Update consistency
  - Read consistency
  - Relaxing consistency
  - Relaxing durability

Update Consistency

- When there are multiple simultaneous updates, we have write-write conflict.
- When storing the data, these updates need to be serialized and the order decided.
  - With multiple servers, "sequential consistency" between nodes is necessary.
- Two approaches for maintaining consistency:
  - Pessimistic, prevent conflicts from occurring — e.g. clients acquire write locks;
  - Optimistic, let conflicts occur and then correct them — e.g. conditional update where clients test the value before updating to check if it changed.
- Alternative optimistic approach:
  - Record all conflicting updates and record that they are in conflict (e.g. version control).
  - Then, the conflicting updates need to be merged. This is highly domain specific.

Update Consistency Tradeoffs

- Immediate reaction is to prefer pessimistic concurrency to avoid conflicts.
- But there is a tradeoff between safety (avoid errors) and responsiveness.
- Pessimistic approaches often degrade responsiveness.
- Also, pessimistic concurrency often leads to deadlocks.

- Using a single node as the target for all writes greatly simplifies maintaining update consistency.

Read Consistency

- Reading non-updated or partially updated data results in read-write conflicts or inconsistent reads.
- Logical consistency — ensuring that different data items make sense together. Dealt in the context of centralized databases.
- In relational databases this is avoided with transactions.
- Aggregate-oriented databases do support atomic updates, but only within a single aggregate.
  - Logical consistency is possible within an aggregate but not between aggregates.
  - In the example, inconsistency would be avoided if [the order, the delivery charge, and the line items] are all part of a single order aggregate.

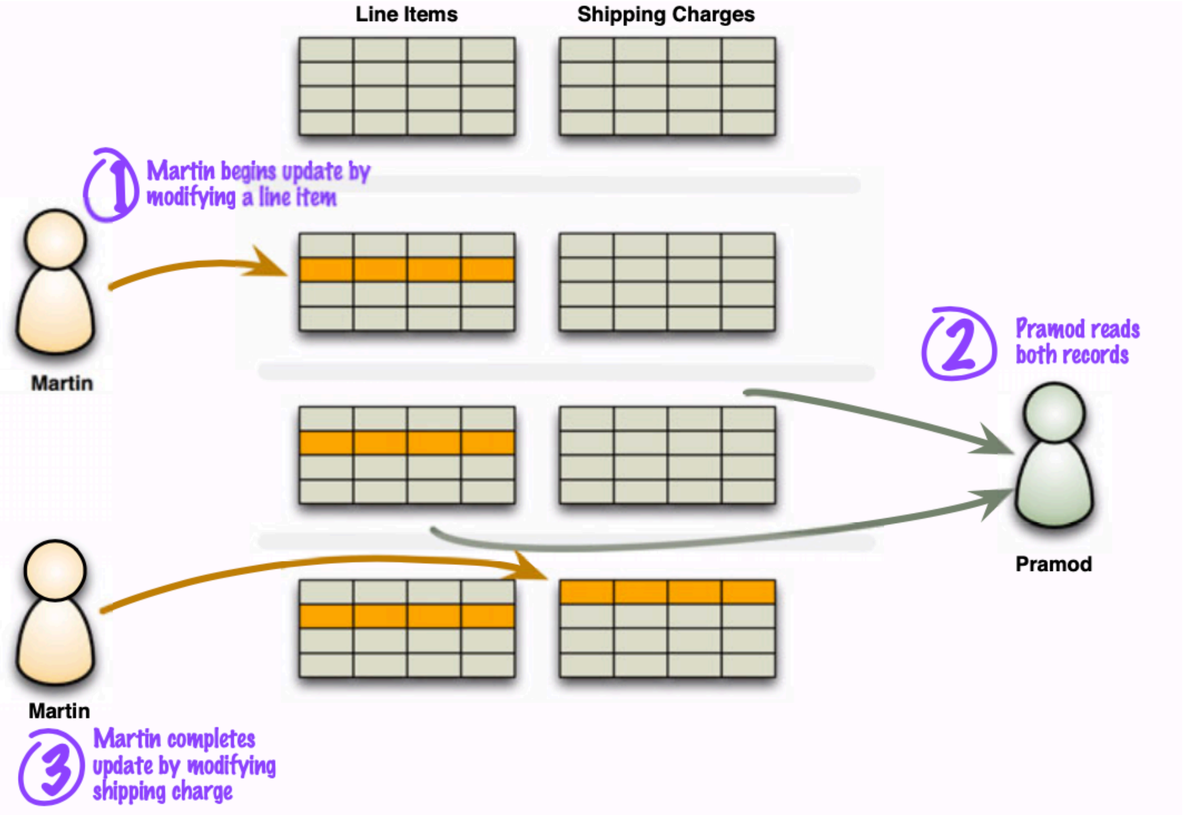

Figure 5.1 *A read-write conflict in logical consistency*

Image from NoSQL Distilled, Sadalage and Fowler (2012)

Replicated Read Consistency

- With replication, new inconsistency problems exist.
- Replication consistency, ensuring that the same data item has the same value when read from different replicas (e.g. geographically distributed updates).
- All updates will eventually propagate fully, this is referred to as "eventually consistent".
- Data out of date is referred to as "stale".

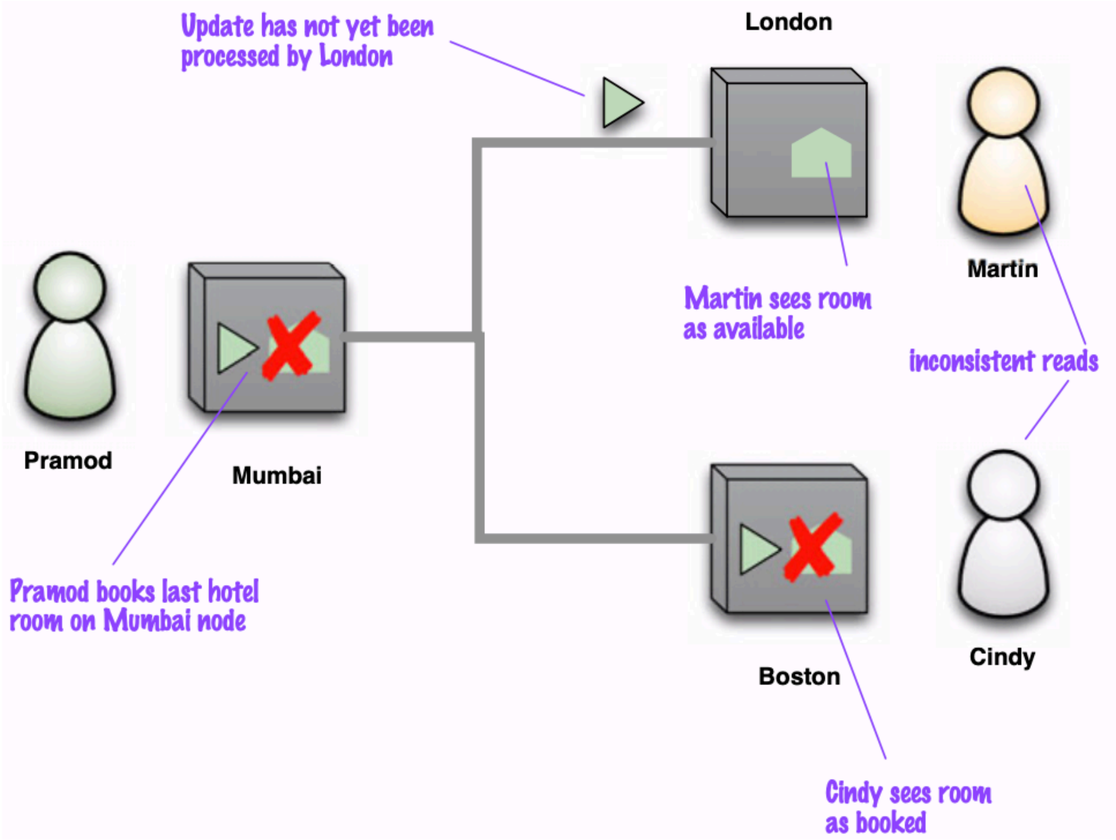

Figure 5.2 *An example of replication inconsistency*

Image from NoSQL Distilled, Sadalage and Fowler (2012)

Session Consistency

- In settings with eventual consistency, a common approach is to provide session consistency — i.e. provide read-your-write consistency within a user's session.
- Session consistency can be implemented with sticky sessions, i.e. sessions that are tied to specific nodes.
- With sticky sessions, as long as read-your-writes consistency is kept on a node, this will be available for sessions too.
- Sticky sessions reduce the ability to load balance the work of nodes.

Relaxing Consistency

- Although consistency is an important characteristic, it often needs to be a tradeoff for other characteristic of the system (e.g. distribution, scale, performance).
- Systems design involves deciding on these tradeoffs.
- In single-server transactional systems, trading off consistency is a familiar concept.
  - The principal tool in this context is the transaction, which can be controlled to relax isolation levels — e.g. read data that hasn't been committed yet, serialize, etc.
- Discarding transactions occurs in practice, e.g. when sharding is introduced, when there are high performance constrains, etc.
  - In these cases, consistency is managed at the business and application level.

The CAP Theorem

- In NoSQL contexts, the CAP theorem is often referred to as the reason why consistency needs to be relax.
- Given Consistency, Availability, and Partition tolerance, a system can only have two.
  - Consistency — every user has an identical view at any given instant.
  - Availability — if a node is accessible, it can read and write data.
  - Partition tolerance — the cluster can survive communication breakages.
- It means that when partitions occur, a tradeoff exists between data availability (or latency) and consistency.
- A single-server system is an example of a CA system, i.e. has Consistency and Availability, but not Partition tolerance (cannot partition, so it not an issue).
- The CAP theorem can be seen as a good starting point to discuss the tradeoffs in systems design, but other issues exist that are not captured (e.g. network delays, failing nodes).

Partition Tolerance

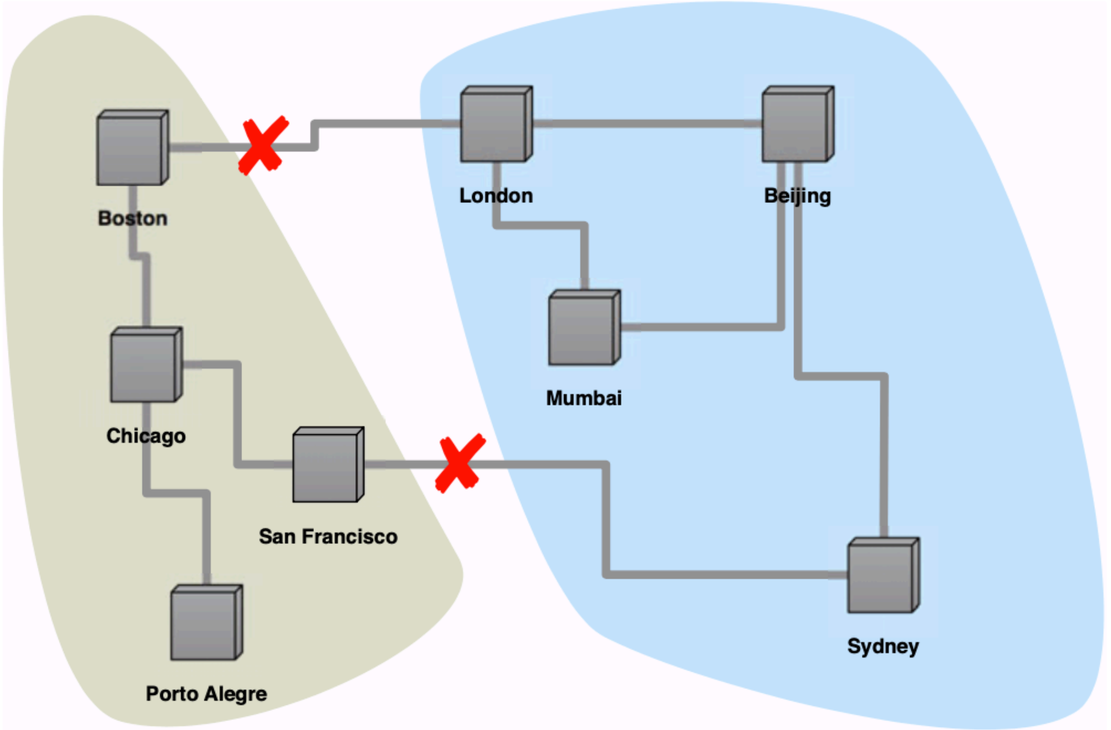

Figure 5.3 *With two breaks in the communication lines, the network partitions into two groups.*

Image from NoSQL Distilled, Sadalage and Fowler (2012)

Relaxing Durability

- The ACID properties of a transaction are: Atomic, Consistent, Isolated and Durable.
- There are cases where durability may be relaxed for higher performance.
- Example: shopping carts in an e-commerce website with many users browsing simultaneously. This scenario is a good candidate for nondurable writes.

Quorums

- Quorums can be used to deal with consistency levels.
- To achieve strong consistency it is not necessary to have acknowledge from all nodes participating in a replication process.
- In write quorums, write consistency can be achieved if the number of nodes participating in the write is more than half of the nodes involved in the replication.
- In read quorums, strong consistent reads can be assured if the number of nodes needed to be contacted for a read (R) plus those confirming a write (W) is greater than the replication factor (N), i.e. $R + W > N$.

Summary of Data Consistency

- Write-write conflicts exist when two clients try to write the same data at the same time. Read-write conflicts occur when one client reads inconsistent data in the middle of another client's write.
- Pessimistic approaches lock data records to prevent conflicts.
  Optimistic approaches detect conflicts and fix them.
- Distributed systems see read-write conflicts due to some nodes having received updates while other nodes have not. Eventual consistency means that at some point the system will become consistent once all the writes have propagated to all the nodes.
- Clients usually want read-your-writes consistency, which means a client can write and then immediately read the new value. This can be difficult if the read and the write happen on different nodes.
- To get good consistency, you need to involve many nodes in data operations, but this increases latency. So you often have to trade off consistency versus latency.

Summary of Data Consistency (2)

- The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency.
- Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.
- You do not need to contact all replicants to preserve strong consistency with replication; you just need a large enough quorum.

Distributed Data Processing

Based on "Chapter 7: Map-Reduce" from NoSQL Distilled, Sadalage and Fowler (2012)

Distributed Data Processing

- Clusters do not only impact data storage, they also impact data processing.
- When dealing with large volumes of distributed data, data processing needs to be handled differently.
- With a centralized database, data processing can be done on:
  - Database server: less programming convenience; higher load on the server; lower data transfers;
  - Client machines: flexible programming environment; higher load on clients; increases data transfers.
- With a distributed system, more computational power exists, but minimizing data transfer across the nodes is essential.
- The goal is to keep as much processing and the data needed together on the same node.

Map-Reduce Pattern

- The map-reduce pattern is a way to organize processing to take advantage of the multiple machines available on a cluster.
- It is inspired in the concept of map and reduce operations in functional programming languages, and was popularized with Google's MapReduce framework.
- A widely used open-source implementation exists as part of the Hadoop project.
- Several databases include their own implementations.

Map-Reduce Process

- A map-reduce process has two core operations:
  - A map operation that takes a single aggregate and outputs several key-value pairs.
  - A reduce operation that takes multiple map outputs with the same key and combines their values.

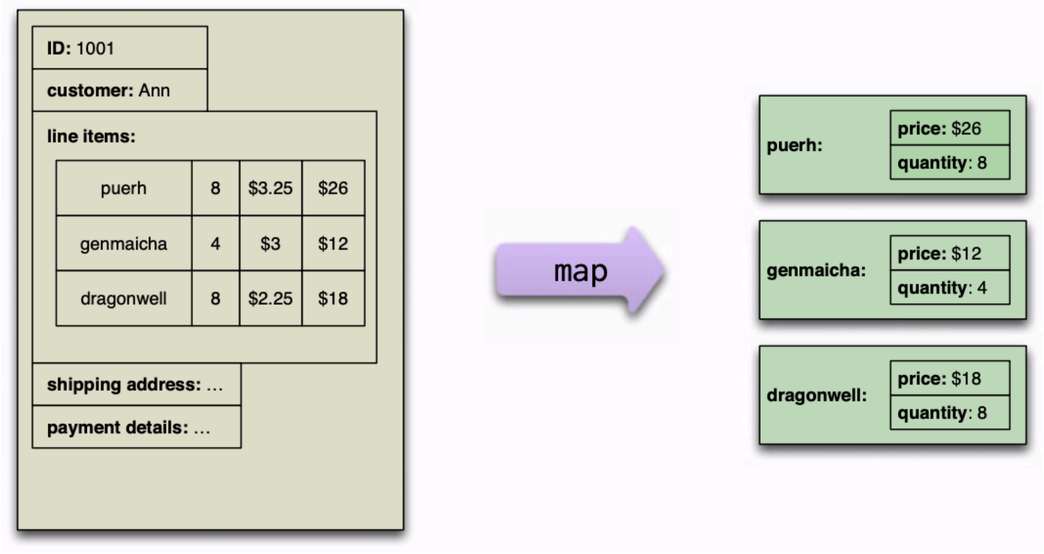

Figure 7.1 *A map function reads records from the database and emits key-value pairs.*

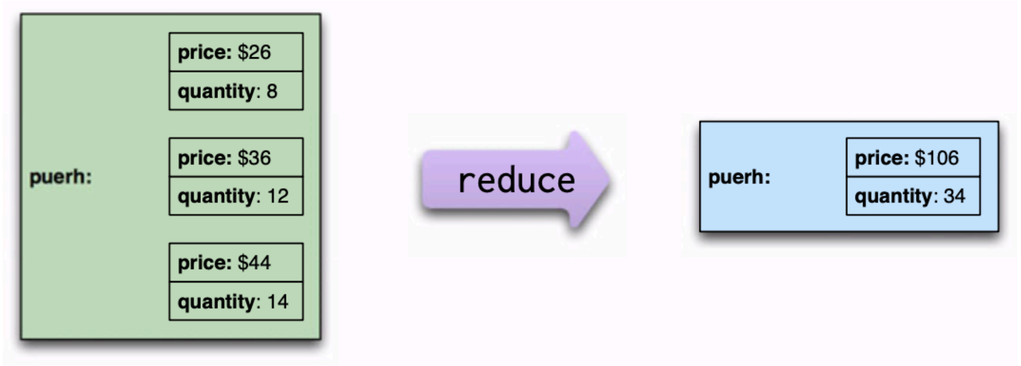

Figure 7.2 *A reduce function takes several key-value pairs with the same key and aggregates them into one.*

Images from NoSQL Distilled, Sadalage and Fowler (2012)

Map-Reduce Process (2)

- The first stage of a map-reduce job is the map operation.
  - A map function takes an aggregate as input and outputs key-value pairs.
  - Each execution of the map function is independent of all the others, thus parallelizable.
  - Map tasks can be executed on each node, gaining parallelism and data locality.
- The second state is the reduce operation.
  - A reduce function takes multiple maps outputs with the same key and combines their values.
  - A reduce task uses values emitted for a single key.
- The map-reduce framework collects all the values for a single pair and calls the reduce function with the key and the collection of all the values for that key.

Partitioning Map-Reduce Jobs

- Map-reduce jobs can be partitioned to allow reduce functions to run in parallel.

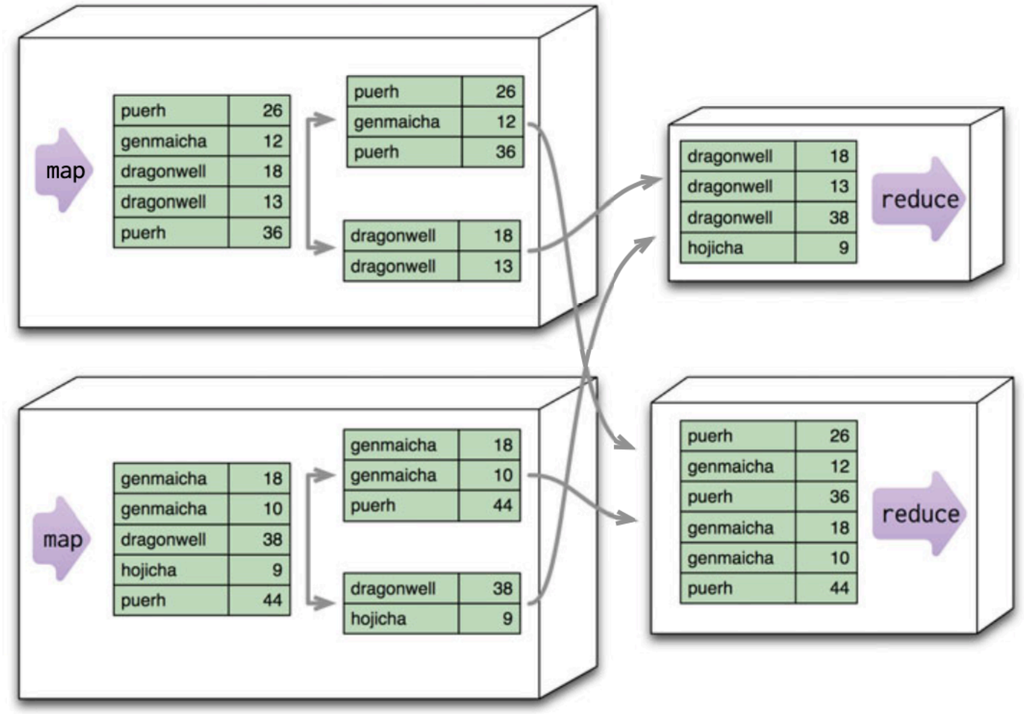

Figure 7.3 *Partitioning allows reduce functions to run in parallel on different keys.*

Image from NoSQL Distilled, Sadalage and Fowler (2012)

Composing Map-Reduce Calculations

- The map-reduce approach trades off flexibility in how data is structured, for simplicity in how data processing is parallelized and executed.
  - Within a map task it is only possible to operate on a single aggregate.
  - Within a reduce task it is only possible to operate on a single key.
- Data structures and algorithms need to be organized to fit in this context.
- Two examples:
  - Calculating averages (which are not composable).
  - Obtaining global counts.

Calculating Averages with Map-Reduce

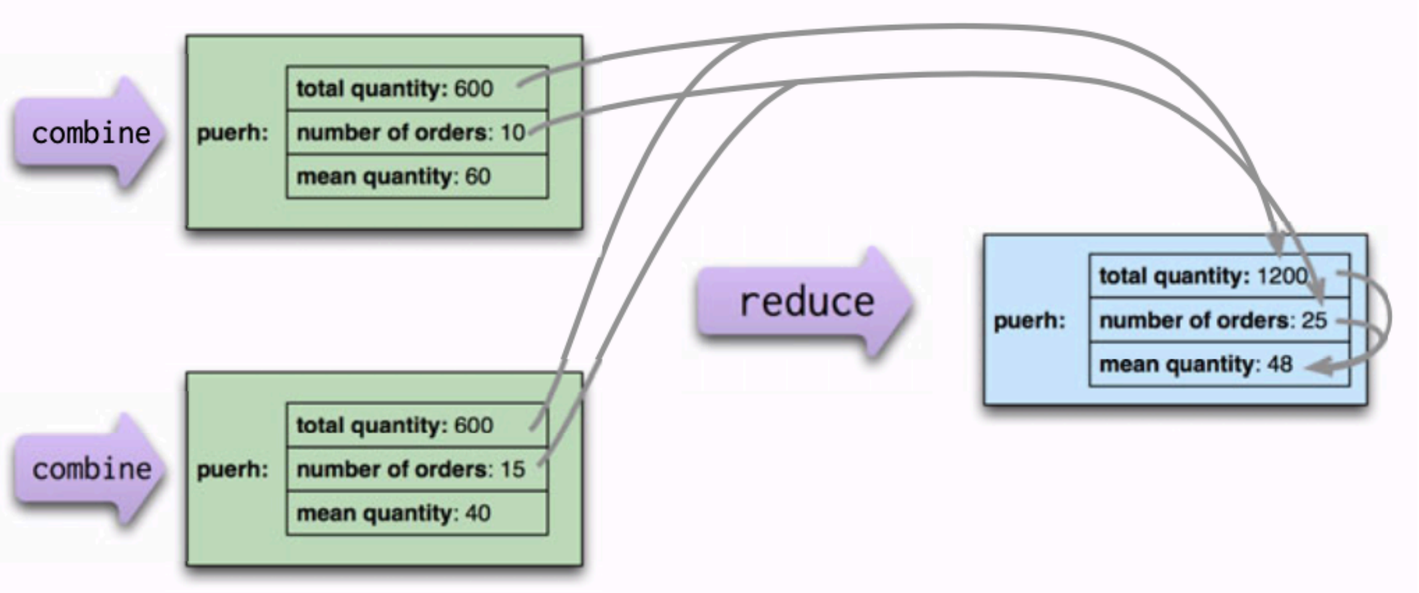

Figure 7.6 *When calculating averages, the sum and count can be combined in the reduce calculation, but the average must be calculated from the combined sum and count.*

Image from NoSQL Distilled, Sadalage and Fowler (2012)

Counting with Map-Reduce

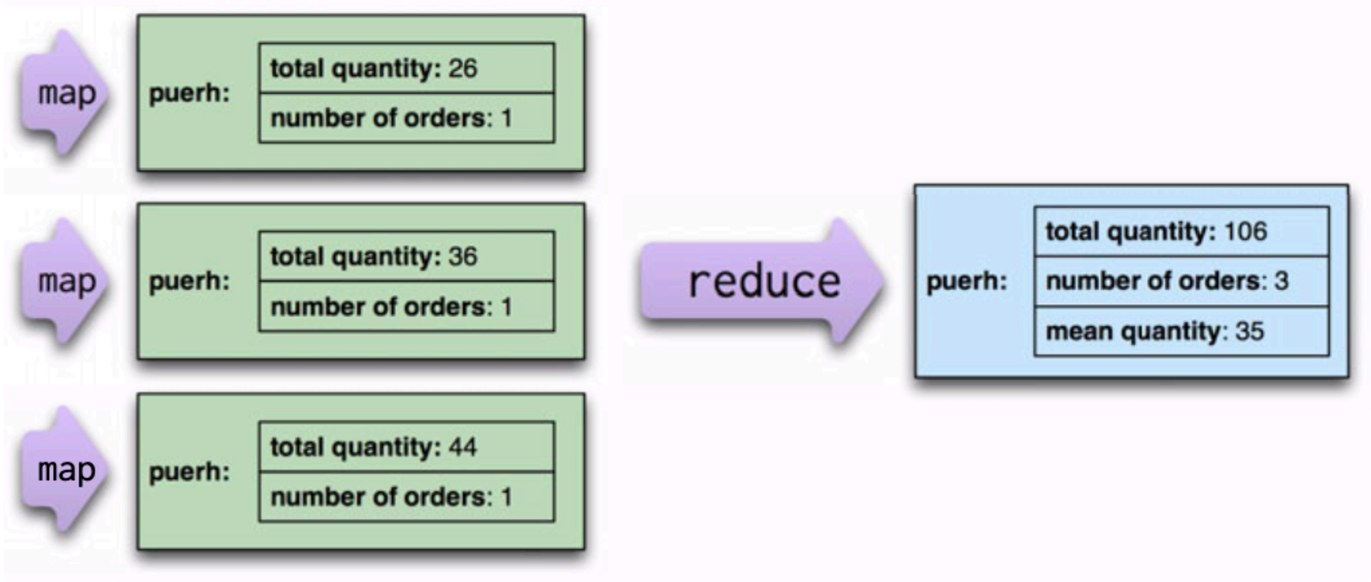

Figure 7.7 *When making a count, each map emits* 1, *which can be summed to get a total.*

Image from NoSQL Distilled, Sadalage and Fowler (2012)

Summary of Distributed Data Processing

- Map-reduce is a pattern to allow computations to be parallelized over a cluster.
- The map task reads data from an aggregate and boils it down to relevant key-value pairs. Maps only read a single record at a time and can thus be parallelized and run on the node that stores the records.
- Reduce tasks take many values for a single key output from map tasks and summarize them into a single output. Each reducer operates on the result of a single key, so it can be parallelized by key.
- Reducers that have the same form for input and output can be combined into pipelines.
  This improves parallelism and reduces the amount of data to be transferred.
- Map-reduce operations can be composed into pipelines where the output of one reduce is the input to another operation's map.
- If the result of a map-reduce computation is widely used, it can be stored as a materialized view.

Summary on Principles of NoSQL Databases

- Relational databases continue to be a very successful technology, as a solution for persistence, for concurrency control, and as an integration mechanism.
- The growth in data volume, the impedance mismatch between application data structures and storage models, and the move to distributed systems integrated through services, all contribute to the emergente of new storage solutions.
- NoSQL solutions are suitable to run on clusters, adopting a distributed architecture.
- The notion of aggregate, i.e. complex unit of data indexed by a key for lookups, is central to running a distributed system.
- Data distribution can be achieved with two main strategies — primary-replica and peer-to-peer.
  This choice impacts storage, network traffic, and availability.
- Distribution impacts consistency and a range of options exists to choose from, which involve different problems and advantages to deal with — update consistency, read consistency, relaxing consistency, and relaxing durability.
- When dealing with distributed data, data processing needs to be handled differently.
  Map-reduce allows computations to be parallelized over a distributed infrastructure and take advantage of data locality.

Questions or comments?

References

- **NoSQL Distilled**
  Pramod J. Sadalage and Martin Fowler
  Addison-Wesley, 2012

- **Next Generation Databases**
  Guy Harrison
  Apress, 2016

- **A Survey on NoSQL Stores**
  Ali Davoudian, Liu Chen, and Mengchi Liu
  ACM Computing Surveys, 2018